GPU Architecture and Programming Model

Jiří Filipovič

Fall 2023

Differences among CUDA GPUs

New generations bring higher performance and new computing capabilities.

- *compute capability* describes richness of GPU instruction set and amount of resources available (registers, number of concurrently running threads, etc.)
- raw performance grows with the number of cores on a GPU

Cards in the same generation differ in performance substantially

- to produce more affordable cards
- due to changes introduces later in the manufacturing process
- to minimize power consumption of mobile GPUs

GPUs Available Today

Currently available GPUs

- compute capability 1.0 - 9.0
  - we will learn the differences later
- 1–108 multiprocessors (19 GFlops - 67 TFLOPs)
- frequency of 800 MHz–1.836 GHz
- width and speed of data bus (64–4096 bit, 6.4–3350 GB/s)

Generations of CUDA GPU

Generations and their computing capability

- Tesla (G80, G90, G200): c.c. 1.0, 1.1, 1.2, 1.3
  - do not confuse with Tesla computing cards
- Fermi (GF100, GF110): c.c. 2.0, 2.1
- Kepler (GK100, GK110): c.c. 3.0, 3.2, 3.5, 3.7
- Maxwell (GM107, GM200): c.c. 5.0, 5.2, 5.3
- Pascal (GP102, GP100): c.c. 6.0, 6.1, 6.2
- Volta (GV100): c.c. 7.0
- Turing (GT100): c.c. 7.5
- Ampere (GA100): c.c. 8.0, 8.6 (GeForce 3xxx)
- Ada Lovelance (AD102): c.c. 8.9 (GeForce 4xxx)
- Hopper (GH100): c.c. 8.9 (Nvidia H100)

Available products

GeForce graphics cards

- mainstream solution for gaming
- cheap, widely used, broad range of performance
- disadvantage – limited memory, limited double precision performance

Professional Quadro graphics cards

- larger memory
- several times more expensive

Tesla

- a solution specially designed for CUDA computing
- offers some HW features not present in GeForce (large memory, double/half precision, NVLink, ECC memory etc.) speeding up some applications
- expensive

GPU Parallelism

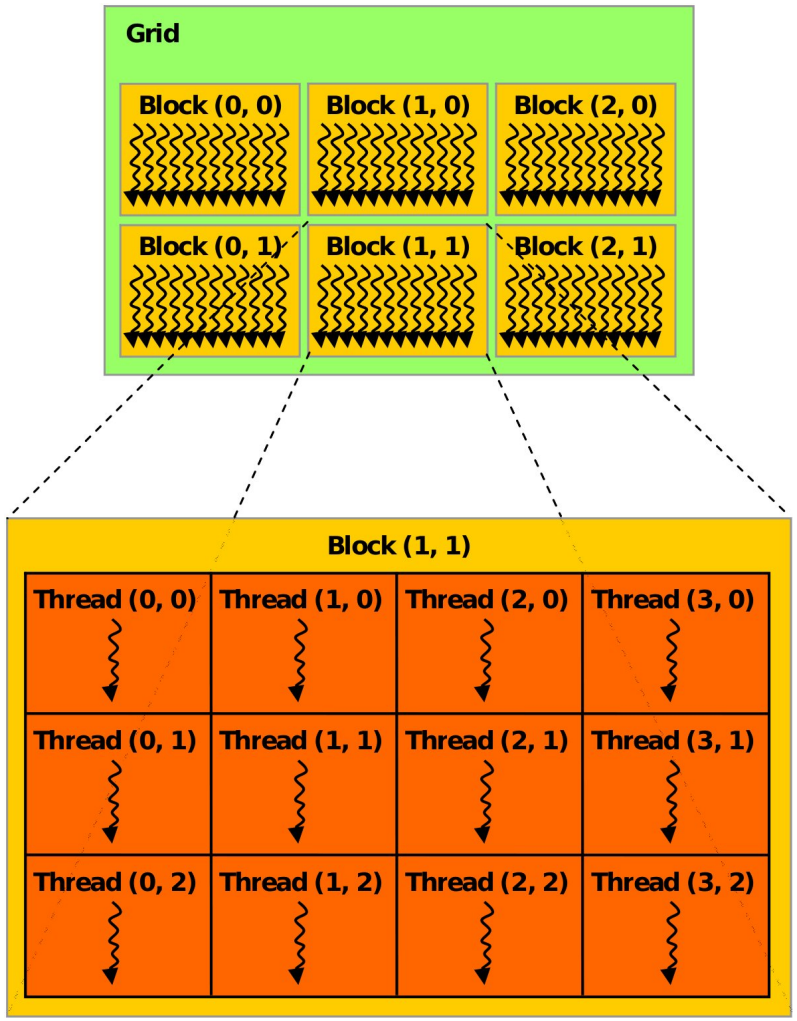
Parallel algorithms need to be designed w.r.t. the parallelism available in the HW

- GPU: array of SIMT multiprocessors working using shared memory

Decomposition for GPU

- coarse-grained decomposition of the problem into the parts that don't need intensive communication
- fine-grained decomposition similar to vectorization (but SIMT is more flexible)

Task Hierarchy



SIMT

A multiprocessor of G80 has one unit executing an instruction

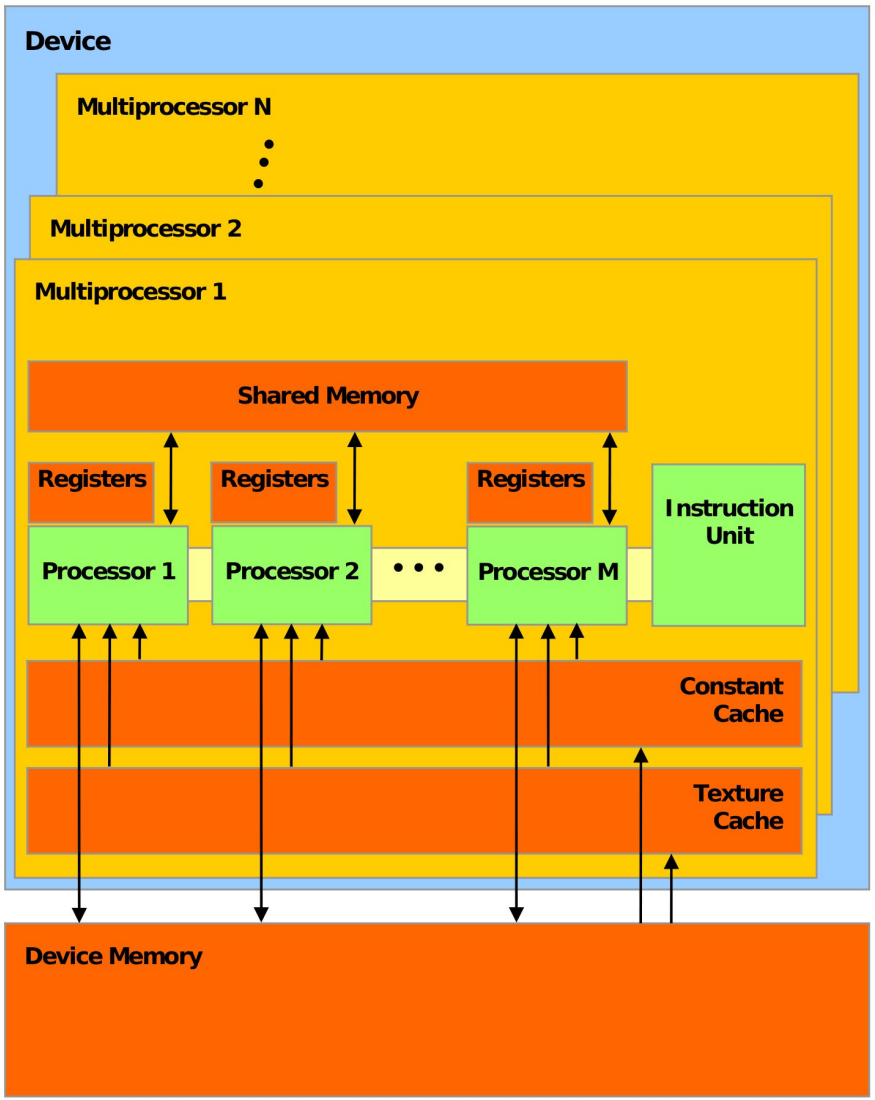
- all 8 SPs have to execute the same instruction
- new instruction is executed every 4 cycles
- 32 threads (so called *warp*) need to execute the same instruction, warp size is fixed for all existing CUDA hardware

How about code branching?

- if different parts of a warp perform different instructions, they are serialized
- decreases performance—should be avoided

The multiprocessor is thus (nearly) MIMD (Multiple-Instruction Multiple-Thread) from programmer's perspective and SIMT (Single-Instruction Multiple-Thread) from performance perspective.

GPU Architecture



SIMT reconvergence

At the end of divergent code, a point of reconvergence is set by the compiler

- creates barrier for threads within the warp
- guarantees threads synchronization after divergent code
- we have to take the reconvergence points in mind – they can create deadlocks, which do not arise in true MIMD
- Volta's and newer GPUs' threads are scheduled independently, thus it can be programmed as a true MIMD processor

SIMT reconvergence

We try to serialize some region of code by the following construct:

```
__shared__ int s = 0;
while (s != threadIdx.x) {};
// serialized region
s++;
```

Thanks to reconvergence point, there is a deadlock (reconvergence point is placed before the incrementation of s).
Fix:

```
__shared__ int s = 0;
while (s < blockDim.x) {
  if (threadIdx.x == s) {
    // serialized region
    s++;
  }
}
```

Thread Properties

GPU threads are very lightweight compared to CPU threads.

- their run time can be very short (even tens of instructions)
- there should be many of them
- they should not use large amount of resources

Threads are aggregated into blocks

- all threads of the block always run on the same multiprocessor (multiple blocks can run at one multiprocessor)
- having sufficient number of blocks is substantial to achieve good scalability

Number of threads and thread blocks per multiprocesor is limited.

Memory Latency Masking

Memory has latency

- global memory has high latency (hundreds of cycles)
- registers and shared memory have read-after-write latency

Memory latency hiding is different from CPU

- no instructions are executed out of order (but ILP can be exploited by forcing finalization of load instruction just before loaded data are needed)
- no or limited cache

When a warp waits for data from memory, another warp may be executed

- allows memory latency hiding
- requires execution of more threads than the number of GPU cores
- thread execution scheduling and switching is implemented directly in HW without overhead

Thread-Local Memory

Registers

- the fastest memory, directly usable in instructions
- local variables in a kernel and variables for intermediate results are placed automatically into the registers
  - if there is sufficient number of registers
  - if the compiler can determine static array indexing
- thread scoped

Thread-Local Memory

Registers

- the fastest memory, directly usable in instructions
- local variables in a kernel and variables for intermediate results are placed automatically into the registers
  - if there is sufficient number of registers
  - if the compiler can determine static array indexing
- thread scoped

Local memory

- data that doesn't fit into the registers go into the local memory
- local memory is stored in DRAM $\Longrightarrow$ slow, high latency
- thread scoped

Shared Memory

Shared memory

- as fast as registers for c. c. 1.x, for newer GPUs little bit slower
  - if memory bank conflicts are avoided
  - instructions can use only one operand in shared memory (otherwise explicit load/store is needed)
- declared using `__shared__` in C for CUDA
- a variable in shared memory can have dynamic size (determined at startup), if declared as `extern` withou size specification
- block scoped

Shared Memory

Static shared memory declaration

```
__shared__ float myArray[128];
```

Dynamic allocation

```
extern __shared__ char myArray[];
float *array1 = (float*)myArray;
int *array2 = (int*)&array1[128];
short *array3 = (short*)&array2[256];
```

It creates an array `array1` of `float` type with size 128, `array2` of `int` type sized 256, and `array3` of floating size. Total size has to be specified at kernel startup.

```
myKernel<<<grid, block, n>>>();
```

Global Memory

Global memory

- an order of magnitude lower bandwidth compared to shared memory
- latency in order of hundreds of GPU cycles
- addressing needs to be coalesced to get optimum performance
- application-scoped
- cached in some architectures, e.g. L1 cache (128 bytes/row) and L2 cache (32 bytes/row) in Fermi architecture

May be dynamically allocated using `cudaMalloc` or statically allocated using `__device__` declaration.

Constant Memory

Constant memory

- read-only
- cached
- cache hit is as fast as registry (under certain constraints), cache miss is as fast as global memory
- limited size (64 kB for GPUs currently available)
- application-scoped

Constant Memory

Declared using `__constant__` keyword; the following function is used for copying data to constant memory:

```
cudaError_t cudaMemcpyToSymbol(const char *symbol,
  const void *src, size_t count, size_t offset,
  enum cudaMemcpyKind kind)
```

Data are copied from system memory (`cudaMemcpyHostToDevice`) or global memory (`cudaMemcpyDeviceToDevice`) from `src` into `symbol`. The copied block has `count` bytes. Copied with `offset` into the `symbol` memory.

Texture Memory

Texture memory

- cached, 2D locality
- read-only for cache coherency reasons
- high latency
- several addressing modes
  - normalization into $[0, 1]$ range
  - truncation or overflowing of coordinates
- possible data filtering
  - linear interpolation or nearest value
- this functionality is "for free" (implemented in HW)

More details are available in CUDA Programming Guide.

Data Cache

Read-only data cache

- c.c. 3.5 or higher
- the same hardware as texture cache (up to Pascal), or shared memory (Volta and newer)
- straightforward usage
- compiler automatically uses data cache, when it recognize that data are read-only
  - we can help with `const` and `__restrict__`
  - usage can be forced by `__ldg()`

System-Local Memory

System RAM

- connected to GPU via PCIe
- CPU (host) and GPU (device) memory transfers are complicated by virtual addressing
- it is possible to allocate so called page-locked memory areas
  - overall system performance may be reduced
  - limited size
  - data are transferred faster over PCIe
  - allows for parallel kernel run and data copying
  - allows for mapping of host address space onto the device
  - allows for `write-combining` access (data are not cached by CPU)

Page Locked Memory

`cudaMallocHost()` is used instead of `malloc()` to allocate the memory; the memory is freed using `cudaFreeHost()`

- `cudaHostAllocPortable` flag ensures page-locked memory for all CPU threads
- `cudaHostAllocWriteCombined` flag turns off caching for CPU allocated memory
- `cudaHostAllocMapped` flag sets host memory mapping in the device address space

Synchronization within the Block

Within block

- native barrier synchronization
  - all threads have to enter it (beware of conditions!)
  - one instruction only, very fast if it doesn't degrade parallelism
  - C for CUDA call **__syncthreads()**
  - Fermi extensions: count, and, or
- shared memory communication
  - threads can exchange data
  - barrier ensures that data are ready
- synchronization latency hiding similar as for memory
  - multiple blocks on multiprocessor

Block Synchronization

Among blocks

- global memory is visible for all blocks
- poor support for synchronization
  - no global barrier for GPUs prior Pascal architecture and CUDA 8.0
  - *atomic operations* on global memory
  - global barrier can be implemented using multiple kernel calls

Matrix Multiplication

We want to multiply matrices A a B and store the result into C.
For sake of simplicity, we only assume matrices sized $n \times n$.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

C language:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Parallelization

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Multiple ways of parallelization

- choose one loop
- choose two loops
- parallelize all the loops

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need tens thousands of threads for good GPU utilization)

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need tens thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need tens thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization using inner loop

- complicated, synchronization needed when writing into C!

Parallelization

Parallelization of one loop

- doesn't scale well, it is necessary to use big matrices (we need tens thousands of threads for good GPU utilization)

Parallelization of two loops

- scales well, number of threads grows quadratically w.r.t. n

Parallelization using inner loop

- complicated, synchronization needed when writing into C!

Best way is thus to parallelize loops over i and j.

First Kernel

We can form the block and grid as 2D array.

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  float tmp = 0;
  for (int k = 0; k < n; k++)
    tmp += A[y*n+k] * B[k*n+x];

  C[y*n + x] = tmp;
}
```

Note similarity to math description – parallel version is more intuitive than the serial one!

Performance

What will be the performance of our implementation?

Performance

What will be the performance of our implementation?
Let's look at GeForce GTX 280

- available 622 GFLOPS for matrix multiplication
- memory bandwidth is 142 GB/s

Performance

What will be the performance of our implementation?
Let's look at GeForce GTX 280

- available 622 GFLOPS for matrix multiplication
- memory bandwidth is 142 GB/s

Flop-to-word ratio of our implementation

- in one step over k, we read 2 floats (one number from A and B) and perform two arithmetic operations
- one arithmetic operation corresponds to transfer of one float
- global memory offers throughput of 35.5 billion floats per second if one warp transfers one float from one matrix and 16 floats from the other matrix, we can achieve 66.8 GFLOPS
- 66.8 GFLOPS is very far from 622 GFLOPS

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?
For computation of one C element, we have to read one row from A and one column from B, that are in the global memory.

How to Improve It?

We hit the limit of global memory. GPUs have faster types of memory, can we use them?
For computation of one C element, we have to read one row from A and one column from B, that are in the global memory.
Is it really necessary to do that separately for each element of C?

- we read the same A row for all the elements in the same row of C
- we read the same B column for all the elements in the same column of C
- we can read some data only once from the global memory into the shared memory and then read them repeatedly from the shared memory

Tiled Algorithm

If we access the matrix in tiles, we can amortize transfers from the global memory:

- we will compute $a \times a$ tile of C matrix
- we read tiles of the same size of matrices A and B into the shared memory iteratively
- the tiles will be multiplied and added to C
- ratio of arithmetic operations to data transfers is a times better

Natural mapping on GPU parallelism

- each tile of C will be computed by a thread block
- shared memory locality ensured
- no inter-block synchronization needed

Tiled Algorithm

How big thread blocks?

- if equal to the tile size, it is limited by the size of shared memory
- limited by the number of threads that can run on GPU
- the reasonable block size is 16×16
  - multiple of warp size
  - one block will have reasonable 256 threads
  - one block needs 2 KB of shared memory
  - the memory will not limit the performance substantially ($16 \cdot 25.5 = 568$ GFLOPS, which is quite close to 622 GFLOPS)

Algorithm

Algorithm schema

- each thread block have tiles `As` and `Bs` in the shared memory
- tiles of A and B matrices will be multiplied iteratively, the results will get accumulated in `Csub` variable
  - threads in a block read tiles into `As` and `Bs` cooperatively
  - each thread mutliplies rows in `As` and columns in `Bs` for its element of `Csub` matrix
- each thread stores one element of the matrix into the matrix C in global memory

Beware of synchronization

- the blocks need to be read completely before the multiplication starts
- before we read new blocks, operation on previous data needs to be completed

Second Kernel

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float As[TILE_SIZE][TILE_SIZE];
  __shared__ float Bs[TILE_SIZE][TILE_SIZE];

  float Csub = 0.0f;
  for (int b = 0; b < n/TILE_SIZE; b++){
    As[ty][tx] = A[(ty + by*TILE_SIZE)*n + b*TILE_SIZE+tx];
    Bs[ty][tx] = B[(ty + b*TILE_SIZE)*n + bx*TILE_SIZE+tx];
    __syncthreads();

    for (int k = 0; k < TILE_SIZE; k++)
      Csub += As[ty][k]*Bs[k][tx];
    __syncthreads();
  }

  C[(ty + by*BLOCK)*n + bx*TILE_SIZE+tx] = Csub;
}
```

Performance

- theoretical limitation of the first kernel is 66.8 GFLOPS, measured performance is 36.6 GFLOPS
- theoretical limitation of the second kernel is 568 GFLOPS, measured performance is 198 GFLOPS
- how to get closer to the maximum performance of the card?
- we need to understand HW and its limitation better and optimize the algorithms accordingly
- topics for the next lectures